Une proposition de composants formels

Pascal Poizat* — **Jean-Claude Royer****

* *LaMI - UMR 8042, Université d'Évry Val d'Essonne*
Tour Évry 2, 523 place des terrasses de l'Agora, F-91000 Évry Cedex
poizat@lami.univ-evry.fr

** *IRIN - Université de Nantes*
2, rue de la Houssinière - BP 92208, F-44322 Nantes Cedex 3
Jean-Claude.Royer@irin.univ-nantes.fr

RÉSUMÉ. *La notion de composant, bien qu'ancienne, n'a pas encore donné toute sa puissance d'expression pour la construction de systèmes logiciels complexes. Les propositions actuelles, au niveau de la conception ou du codage, bien que démonstratives nous semblent encore très insuffisantes. Il est fondamental que la notion de composants logiciels apparaisse le plus tôt possible dans le cycle de vie, et en particulier au niveau spécification formelle. Dans cet article nous présentons notre notion de composant formel sur une étude de cas modélisée en* KORRIGAN. *Nos composants sont basés sur une notion de vue graphique et textuelle et utilisent des systèmes de transitions symboliques, des spécifications algébriques et une colle temporelle. Nous illustrons divers aspects, notamment la composition, la réutilisation et la communication entre composants.*

ABSTRACT. *The concept of component, even if an old one, has not yet given all its expressive power as far as building complex systems is concerned. New proposals for components have appeared at design or code level. Although they are demonstrative, we think they are still not sufficient enough. It is fundamental for us that the concept of sofware component appears as soon as possible in the life cycle, in particular at the formal specification level. In this article we present our formal component concept on a case study modelled with* KORRIGAN. *Our components are based upon textual and graphical views, using symbolic transition systems, algebraic specifications and a temporal glue. We illustrate some of their features, amongst them composition, reuse and communication between components.*

MOTS-CLÉS : *composants formels, composition, réutilisation, patrons de communication, vues.*

KEYWORDS: *formal components, composition, reuse, communication patterns, vues.*

1. Introduction

Pour concevoir des architectures logicielles complexes nous avons besoin d'une approche structurée qui nous permette de décomposer un système en parties et de le recomposer à partir de ces parties. Les projets de Génie Logiciel, pour être opérationnels, doivent disposer de techniques adaptées à la grande taille des systèmes réels. Mais tout d'abord nous devons disposer de techniques efficaces au niveau composant. La notion de composant est ancienne (module, type abstrait, cluster), l'idée était déjà présente dans Simula. Elle a aussi été étudiée dans le cadre du langage Modula et des composants logiciels de Booch [BOO 87]. Récemment des propositions d'assez bas niveau (EJB, DCOM, UML-RT [SEL 98]) ont fait irruption. Nous nous plaçons plutôt ici à un niveau abstrait et formel, comme WRIGHT [ALL 97], car cela nous semble fondamental, entre autres pour une bonne réutilisation.

Nous supposons que les systèmes sont décomposables en un arbre de composants séquentiels (feuilles) et de composants concurrents (nœuds) comme dans les algèbres de processus ou en LOTOS [ISO 89]. Informellement un composant est une unité d'information composable et réutilisable et dont on connaît un contrat, une interface ou des conditions d'utilisation. À strictement parler, nous nous intéressons plutôt à des types de composant qu'à des composants. Dans le domaine de la programmation séquentielle on connaît assez bien les types abstraits et les classes que l'on peut considérer comme des types de composants. L'interface des composants est souvent réduite à la signature mais on sait depuis [HOA 72] qu'une approche avec des pré-postconditions ou des axiomes est bien meilleure. Dans le domaine concurrent, la notion de composant est plutôt l'acteur, la tâche ou le processus. Dans ce cas l'interface est un protocole, des portes ou des canaux de communication.

Il est également reconnu que plusieurs points de vue sont utiles pour spécifier un système, c'est pourquoi nous devons proposer des *composants mixtes*. Dans la littérature, particulièrement dans le domaine des spécifications formelles, on trouve plusieurs termes voisins, composant hétérogène ou intégration d'aspects. Dans cet article un composant mixte est une unité d'information comportant des aspects fonctionnels ou statiques (types de données, opérations), dynamiques (contrôle, communication, synchronisations) et architecturaux (compositions, liens entres les composants). Le besoin d'utiliser et de combiner des aspects différents d'un système a déjà été évoqué dans des travaux antérieurs, par exemple dans les approches combinant les types de données avec d'autres formalismes (*e.g.* LOTOS [ISO 89], SDL [ELL 97] ou [SMI 97, FIS 97]). Ce besoin est à la base de la programmation par aspects [KIC 97]. Il est aussi présent dans les approches orientées-objets comme UML [CON 99] où les aspects statiques et dynamiques sont présentés dans différents diagrammes (diagramme de classe, diagramme d'interaction, Statecharts). Toutefois, le lien (formel) et la cohérence entre ces différents aspects ne sont généralement pas définis et ils ne sont pas triviaux. Un composant doit être accompagné de son mode d'emploi : interface, portes, canaux, préconditions, axiomes, etc. L'interface des composants est d'une importance cruciale car le producteur du composant n'est pas forcément son utilisateur.

Il est important que ce type d'information soit formel pour abstraire, simplifier et éviter les ambiguïtés. Dans l'approche présentée ici nous considérons que la dynamique prime sur la statique, c'est-à-dire que l'interface d'un composant est en fait composée de points de synchronisation, de communications et de gardes. La partie statique, si elle est nécessaire, sera encapsulée dans la dynamique. Il est important d'abstraire le plus possible la partie colle entre les composants : elle doit permettre d'exprimer les synchronisations, les communications et les liens entre les gardes tout en restant lisible. Un dernier point est que ces composants doivent avoir à la fois une description textuelle et graphique pour faciliter leur utilisation et la compréhension des architectures. Notre modèle, KORRIGAN [POI 00], est dévolu à la spécification formelle et structurée de composants mixtes réutilisables. Cette approche permet de garder les avantages des langages spécialisés dans les divers aspects (*i.e.* système de transitions symboliques pour les comportements dynamiques, spécifications algébriques pour les types de données et une colle à base d'axiomes et de logique temporelle pour la composition) tout en permettant d'avoir un cadre global avec une sémantique uniforme.

Le contenu de cet article est le suivant. La section 2 décrit le cahier des charges du service de téléphonie. La section 3 présente la modélisation en KORRIGAN de cette étude de cas. Nous illustrons la définition des aspects statiques et dynamiques des composants, la structuration et l'expression des communications dans un système complexe ainsi que divers cas de réutilisation. La section 4 présente une comparaison avec les travaux voisins et une conclusion résume les principaux points de notre approche.

2. Le système de téléphonie

Il s'agit d'un système de téléphonie simplifié servant à illustrer notre propos. La description complète de l'étude de cas se trouve dans [POI 00]. Bien qu'elles puissent parfois paraître un peu académiques, nous pensons que ses caractéristiques sont assez significatives de cas réels. Ce système doit permettre à des clients de communiquer via un serveur central.

2.1. *Les clients simples ou abonnés*

Un *client* ou utilisateur est identifié par un numéro personnel à dix chiffres. Un client peut avoir deux statuts : il peut être simple ou abonné. Lorsqu'une communication se met en place entre deux clients, l'*appelant* est celui qui a demandé la communication et l'*appelé* est l'autre client. Les activités des clients dépendent du fait qu'ils soient simples ou abonnés. Un *appelé potentiel* est un client pouvant être appelé par un appelant abonné.

Une communication entre un appelant i et un appelé j est possible quand l'appelant est abonné (mais ce n'est pas nécessaire pour l'appelé). Nous devons également avoir $i \neq j$ et j doit être enregistré en tant qu'appelé potientiel de i. De plus une commu-

nication entre i et j ne doit pas être déjà en cours et i a préalablement demandé une communication avec j qui l'a acceptée.

Le client simple peut, lorsqu'il n'est pas en communication, s'abonner auprès du serveur. Il peut être appelé par le serveur lorsqu'un appelant abonné en fait la demande. Il peut refuser ou accepter une communication en provenance d'un appelant abonné. Il peut interrompre une communication en cours.

Un client abonné a les mêmes droits qu'un client simple. Mais en plus il peut, lorsqu'il n'est pas en cours ou en demande de communication, demander au serveur d'annuler son abonnement. Il peut demander au serveur d'enregistrer un nouveau client comme l'un de ses appelés potentiels. Il peut demander au serveur de supprimer un client de la liste de ses appelés potentiels. Il peut demander au serveur une communication avec un autre client.

2.2. *Le serveur central*

Le serveur central est composé de quatre unités : une unité de connexion, une unité de gestion, une unité de déconnexion et une base de données.

L'*unité de connexion* s'occupe des demandes de connexion. Elle envoie une requête à l'appelé pour lui signaler qu'une demande de connexion a été faite. Elle établit la connexion entre l'appelant et l'appelé. L'*unité de gestion* gère toutes les demandes de changement de statut (client simple / client abonné). Elle gère les demandes de mise à jour de la liste des appelés potentiels des clients abonnés. L'*unité de déconnexion* s'occupe des demandes de déconnexion et elle peut décider d'elle-même d'interrompre une connexion. L'*unité de base de données* contient l'ensemble des numéros d'abonnés, et pour chaque abonné l'ensemble des numéros de ses appelés potentiels. Elle prend en compte les changements de statut des clients, et toute modification éventuelle des listes d'appelés potentiels. Elle dispose d'informations sur les communications en cours, l'appelant et l'appelé.

3. Les composants en KORRIGAN

Nous nous intéressons à des systèmes mixtes et nous avons donc besoin de décrire différents aspects, notamment statiques, fonctionnels, dynamiques et architecturaux. Ces différents aspects sont décrits par des constructions textuelles ou graphiques appelées *vues*.

La hiérarchie des vues (Fig. 1) est un peu complexe en KORRIGAN du fait que l'on cherche à spécifier et structurer des systèmes réalistes. Nous ne décrirons pas ici tous les éléments mais nous introduirons ceux qui nous sont nécessaires au fur et à mesure. Une vue statique sert à décrire des types de données et une vue dynamique décrit un comportement dynamique séquentiel. Une vue d'intégration permet de coller ensemble une vue statique et une vue dynamique. La vue de composition

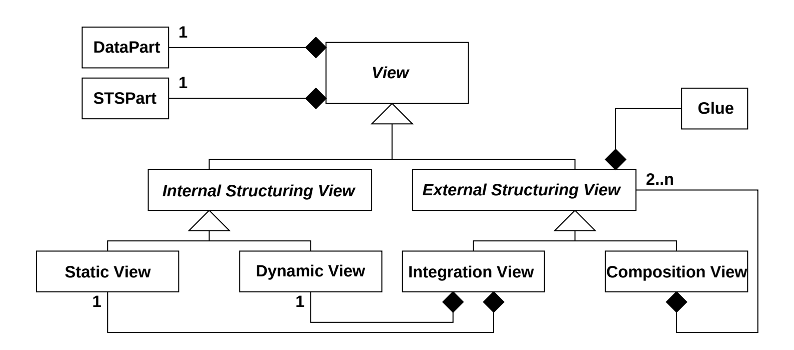


Figure 1. *Hiérarchie (notations UML) des vues de* KORRIGAN

permet de définir des systèmes concurrents composés de plusieurs vues (d'intégration ou de composition). Une description plus détaillée, ainsi que les notations du modèle KORRIGAN, qui sont inspirées par UML, LOTOS et SDL, pourront être trouvées dans [CHO 01a].

KORRIGAN dispose de notations pour décrire les types de données de façon abstraite. Ces notations sont celles habituellement utilisées dans le domaine des spécifications algébriques. Ce qui est plus original est l'utilisation des systèmes de transitions symboliques et la façon dont des aspects différents peuvent être collés, par exemple un type de données avec un objet dynamique.

3.1. *Interfaces dynamiques*

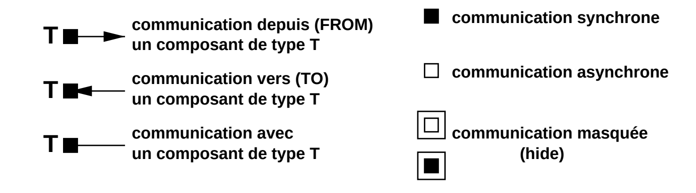


Figure 2. *Notations pour les interfaces dynamiques*

La figure 2 décrit les notations utilisées pour représenter graphiquement certaines interfaces dynamiques des composants parmi les plus courantes. Un carré (blanc ou noir suivant le type de communication) est un service du composant. Il est étiqueté par le nom du service et les communications associées.

3.2. *Système de transitions symbolique*

Pour des raisons de lisibilité, la description d'un comportement dynamique se fait à l'aide d'un *système de transitions symbolique* (STS). Le comportement dynamique d'un utilisateur de base est donnée dans la figure 3.

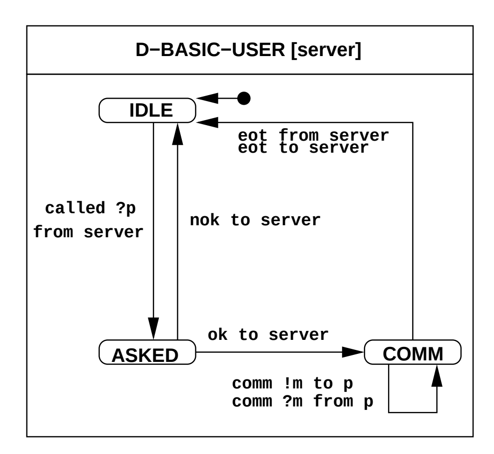


Figure 3. *Le STS d'un utilisateur de base*

Il s'agit d'une originalité notable par rapport à beaucoup d'approches formelles qui n'utilisent que des systèmes de transitions étiquetés. Contrairement aux systèmes étiquetés, les systèmes symboliques disposent de variables et de gardes sur leurs transitions. Dans cette figure nous avons trois états `IDLE, ASKED, COMM` et des transitions étiquetées par des interfaces dynamiques. Une étiquette comme `called ?p from server` signifie un appel avec réception d'une valeur sur la variable `p` en provenance du composant `server`. La notation pour une émission est `!m` où `m` est une expression relative au composant courant. Une telle spécification est générique car `serveur` est une variable qui sert à l'envoi (`from server`) ou à la réception (`to server`). La notation des STS est assez proche des Statecharts mais elle a comme principal avantage une connexion précise avec les types de données sous-jacents au composant. Les STS sont un bon moyen de décrire des systèmes dynamiques (notamment quand leur taille n'est pas bornée) de façon concise et très lisible. Les STS ont une représentation textuelle, voir la figure 4.

3.3. *La notion de vue*

Une *vue* est l'association d'un type de données, d'une fonction d'abstraction et d'une interface dynamique (aussi appelé comportement). C'est la fonction d'abstraction qui réalise la cohérence entre la partie données et la partie dynamique. Cette fonction est décrite par un langage de pré-post conditions. Nous avons une liaison forte entre les trois éléments d'une vue. L'abstraction définit une relation d'équiva-

DYNAMIC VIEW D-BASIC-USER	
SPECIFICATION	**STS**
imports PId, MSG	• → IDLE
generic on	IDLE − called ?p :PIdUser **from** server → ASKED
server :PIdServer	ASKED − ok **to** server → COMM
ops	ASKED − nok **to** server → IDLE
called ?p :PIdUser **from** server	COMM − comm ?m :Msg **from** p → COMM
ok **to** server	COMM − comm !m :Msg **to** p → COMM
nok **to** server	COMM − eot **to** server → IDLE
comm ?m :Msg **from** PIdUser	COMM − eot **from** server → IDLE
comm !m :Msg **to** PIdUser	
eot **to** server	
eot **from** server	

Figure 4. *La vue textuelle dynamique d'un utilisateur de base*

lence partielle sur le type de données. Elle est pour des raisons de lisibilité représentée graphiquement par un STS.

Toutes les vues disposant d'un STS il devient alors assez facile de coller, en définissant des points de synchronisation, deux vues de nature quelconque. Il existe un type particulier de vue, les *vues globales*, dont la sémantique est opérationnelle et basée sur les systèmes de transitions et la réécriture de termes. La sémantique des autres vues est obtenue en les ramenant à une vue globale. Cette correspondance est relativement simple pour un composant statique ou pour un composant dynamique séquentiel. C'est un peu plus complexe dans le cas d'intégrations d'aspects de composants ou de composition concurrentes. Ceci se fait en calculant des parties données, abstraction et comportement globales. Un tel calcul a été décrit dans [CHO 00].

3.4. *Schéma d'architecture*

L'architecture du système est représenté par un schéma du même nom (Fig. 5) qui décrit la composition globale du système. Le système de téléphonie est composé de clients et d'un serveur. Il est générique sur le nombre `N` de clients. Les notations utilisées dans ce schéma sont volontairement voisines de celles d'UML (compositions, héritage, généricité), voir [CHO 01a] pour une justification de ces choix. La structure du serveur est une simple composition de quatre composants qui sont : `UC` (l'unité de connexion), `UDB` (l'unité de base de données), `UD` (l'unité de déconnexion) et `UM` (l'unité de gestion). Un composant en KORRIGAN est généralement décrit par une vue graphique pour des raisons de convivialité. Mais il possède toujours une description textuelle et formelle ce qui est indispensable pour les traitements automatiques.

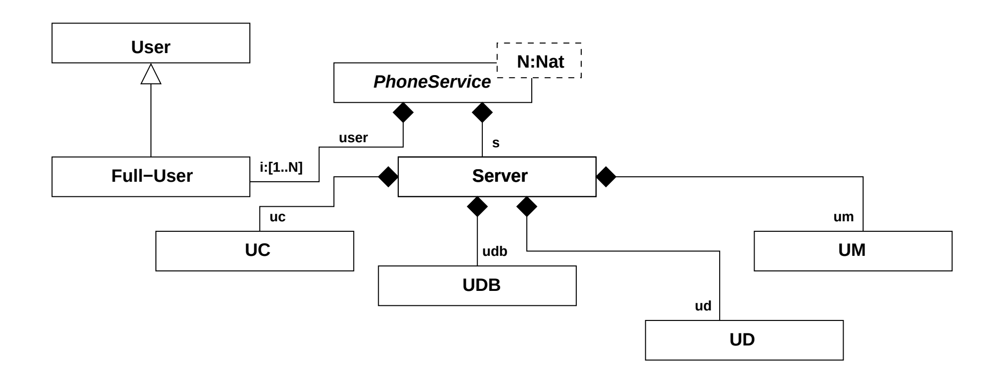


Figure 5. *L’architecture du système de téléphonie*

3.5. *Une forme simple d’héritage*

Généralement l’héritage [MEY 88] permet d’ajouter de nouvelles méthodes dans une classe et autorise la surcharge et le masquage. Il peut également être utilisé pour ajouter des contraintes. L’héritage de comportement statique est délicat et celui de comportement dynamique encore plus [MES 93, PAP 97].

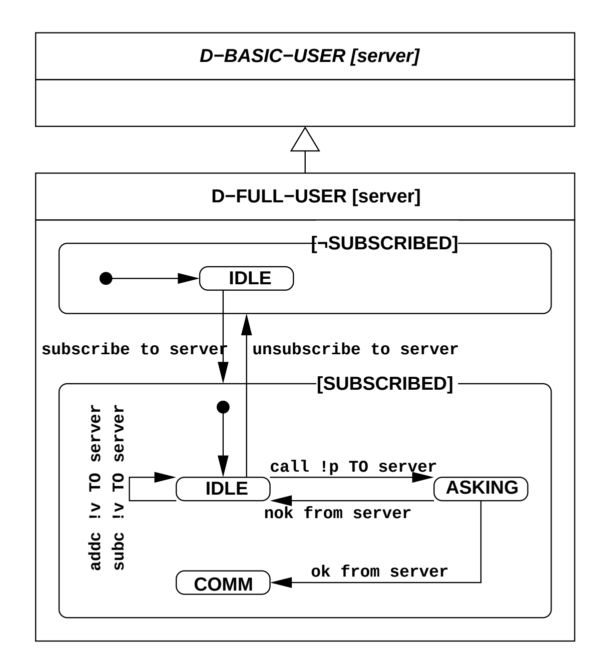


Figure 6. *Le STS du client*

Nous autorisons ici une forme simple d'héritage entre vues du même type (statique ou dynamique), par exemple ici entre deux vues dynamiques (Fig. 6). Notre sémantique de l'héritage est restreinte à l'addition de nouvelles conditions (dans la partie abstraction de la vue) et de nouvelles opérations dans la sous-vue. Il ne permet pas la surcharge et il n'y a pas de masquage car cela occasionne actuellement trop de difficultés pour le spécifieur au niveau de l'écriture des axiomes.

Dans notre étude de cas nous pouvons définir des utilisateurs de base qui peuvent uniquement être appelés. Les clients abonnés peuvent en appeler d'autres et s'abonner ou non à ce nouveau service. Les clients simples peuvent s'abonner et devenir des clients abonnés. Pour définir un client on peut dessiner un nouveau STS *from scratch* mais une meilleure solution est de construire cette vue dynamique par héritage de la vue dynamique de l'utilisateur de base (Fig. 6).

Le fait d'être abonné ou client simple est dénoté par la nouvelle condition `SUBSCRIBED`. Dans une vue héritant d'une autre (on parle de sous- et de super-vue) de nouveaux états, correspondants à de nouvelles conditions apparaissent. La représentation graphique de la sous-vue est simplifiée pour représenter les nouveaux états et les nouvelles transitions. Les états avec le même nom sont superposés (*e.g.* `IDLE` et `COMM` dans la figure 6 qui proviennent de la super-vue). Un super-état est associé à chaque combinaison des valeurs des nouvelles conditions. Cette vue a deux super états `SUBSCRIBED` et ¬`SUBSCRIBED`. Ces états sont hiérarchiques puisque chacun inclue le STS complet de la vue héritée. Le STS réel est défini en utilisant la hiérarchie des états pour chaque valeur des nouvelles conditions. La notation préfixe est utilisée ici, par exemple `SUBSCRIBED.IDLE` désigne l'état `IDLE` du super état `SUBSCRIBED`.

3.6. *Généricité*

Les types génériques auxquels nous sommes habitués sont des listes, des ensembles, bref des structures de données classiques. Ici bien sûr les vues statiques peuvent être génériques mais également les vues dynamiques.

La vue dynamique de la base de données (Fig. 7) comporte trois variables dénotant les identifiants des unités qui sont en communication avec elle. La vue dynamique décrit le protocole de communication avec la base de données qui gère les clients, le type de service et les communications en cours. Il faut pouvoir décrire les effets des événements dynamiques du STS sur le type de données interne à la base. Ceci est fait quand la vue statique et la vue dynamique d'un composant sont intégrées dans une vue dite d'*intégration*.

Une vue d'intégration est une construction KORRIGAN destinée à assembler l'aspect statique et l'aspect dynamique d'un composant. Elle permet, en quelque sorte, de définir des composants *mixtes* qui serviront de base dans des compositions concurrentes. La figure 8 donne la vue textuelle d'intégration de l'unité de base de données en KORRIGAN. Une vue d'intégration graphique utilise les mêmes notations qu'une vue de composition graphique (voir Fig. 9).

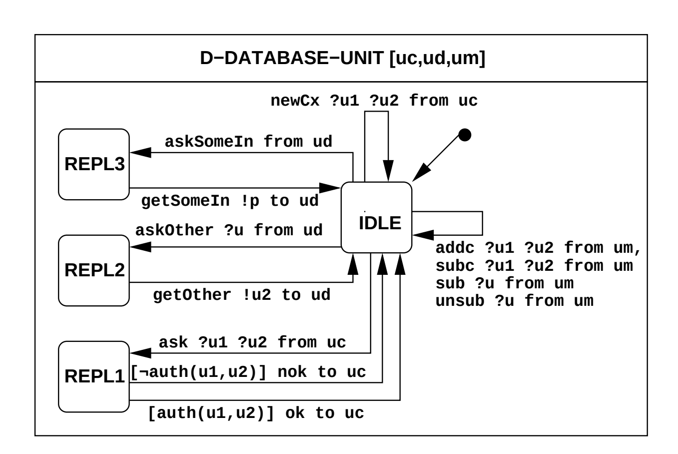


Figure 7. *Le STS de l'unité de base de données*

INTEGRATION VIEW DATABASE-UNIT (UDB)

SPECIFICATION

generic on
uc :PIdUC, ud :PIdUD, um :PIdUM

COMPOSITION

is
sl : SET [PIdFullUser]
al : SET [Couple [PIdUser]]
cl : SET [Couple [PIdUser]]
d : D-DATABASE-UNIT [uc,ud,um]
axioms {al.member((u1,u2)) == d.auth(u1,u2)}
with true, {
(sl.add(u), d.sub ?u **from** um)
(sl.remove(u), d.unsub ?u **from** um)
(al.add((u1,u2)), d.addc ?u1 ?u2 **from** um)
(al.remove((u1,u2)), d.subc ?u1 ?u2 **from** um)
(cl.add((u1,u2)), d.newCx ?u1 ?u2 **from** uc)
(cl.member(u, u2), d.getSomeIn !u **to** ud ∨ d.getSomeIn !u2 **to** ud)
(cl.member((u,u2)) ∨ cl.member((u2,u)),
d.(− askOther ?u **from** ud > − getOther !u2 **to** ud >))}
initially true

Figure 8. *La vue d'intégration de l'unité de base de données*

La partie statique est composée de trois vues statiques, ce qui est un raccourci syntaxique par rapport au diagramme des vues de la figure 1. Les trois vues (`sl, al` et `cl`, Fig. 8) sont obtenues par instanciation de la vue statique du type `SET` et utilisées pour

représenter l'organisation des données interne à la base. `sl` est l'ensemble des identifiants des clients ayant souscrits au service d'appel. `al` correspond aux autorisations d'appel des clients, si un couple `(u1,u2)` est dans `al` alors `u2` est un appelé autorisé pour l'appelant `u1`. `cl` contient les couples de clients en cours de communication.

La colle entre composants est un ensemble d'axiomes et de règles temporelles reliant des événements de chaque composants.

La clause `axioms` exprime la correspondance entre les gardes de la vue dynamique (Fig. 7) et les opérations de la vue statique. Le seul axiome de cette clause dit que la garde `auth` de la figure 7 correspond à l'opération `member` des autorisations d'appel.

La clause `with` de la vue d'intégration est utilisée pour synchroniser les événements dynamiques définis dans le protocole de la base (son comportement dynamique) avec les opérations sur les ensembles qui réalisent le type de données (la vue statique). Chaque couple de cette clause exprime une synchronisation soit entre états (premier argument du `with`) soit entre transitions (deuxième argument). Par exemple, le couple `(cl.add((u1,u2)), d.newCx ?u1 ?u2` **from** `uc)` énonce que si un événement `newCx ?u1 ?u2` est reçu de l'`uc` (une demande de connexion) dans la vue dynamique de la base alors l'opération `add` est effectuée sur l'élément `cl` (les clients en communication) de la vue statique avec comme arguments `u1` et `u2`. Les clauses de collage utilise une logique temporelle quantifiée implicitement par l'opérateur AG signifiant *à tout instant logique* et qui permet de coller des états ou des transitions. Le dernier couple de cette clause permet d'exprimer le fait que si une demande `askOther` il y a synchronisation avec l'action `member` et immédiatement après une réponse `getOther` est faite.

Finalement la clause `initially` est utilisée pour définir des états initiaux particuliers de la vue.

3.7. *Communications et patrons*

Comme dans les langages normalisés pour la spécification des protocoles et dans les langages à objets, des idiomes ou patrons peuvent être utilisés pour décrire des situations générales ou des architectures courantes. Par exemple ici le service de téléphonie est du type client-serveur, un tel patron est représenté par la figure 9. La colle utilisée dans les vues de composition est la même que celle des vues d'intégration. La clause `ALONE` désigne le type de synchronisation utilisé par le composant, *i.e.* les couples d'actions non explicitement synchrones sont asynchrones (comme en LOTOS).

La sémantique des communications dans le patron client-serveur est expliquée dans la figure 10. Le premier cas est une diffusion du serveur vers les n clients et le deuxième une synchronisation point à point entre le serveur et les n clients.

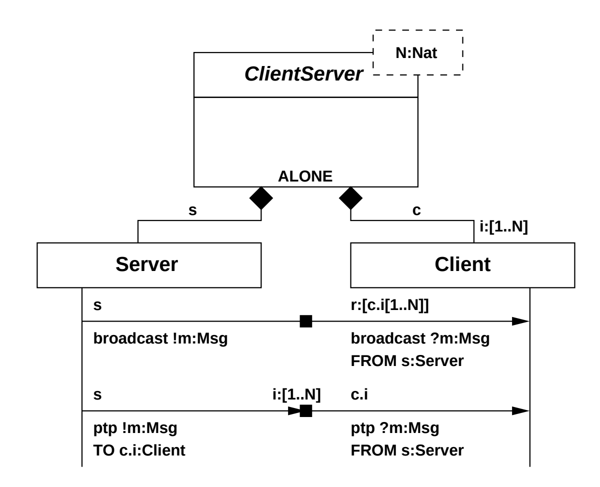


Figure 9. *Le patron du client-serveur en* KORRIGAN

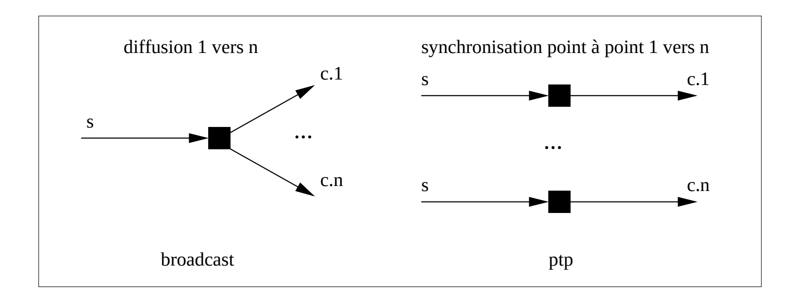


Figure 10. *Abbréviations de communication*

En appliquant le patron précédent à notre cas nous obtenons le squelette de l'architecture du système avec ses communications. Celui-ci a ensuite été adapté et complété pour nos besoins spécifiques mais n'est pas présenté ici faute de place.

4. Comparaison avec d'autres travaux

Le sujet est d'actualité et beaucoup de travaux sont concernés par l'approche composants. Nous allons nous limiter à ceux qui nous semblent les plus pertinents ici.

Une approche très voisine et antérieure est celle de WRIGHT [ALL 97]. Le cadre et les préoccupations sont les mêmes que les nôtres. WRIGHT se base sur CSP et peut en fait être vue comme un habillage de ce langage. Ceci a l'avantage de permettre directement des vérifications en utilisant la vérification de modèle. Notre approche propose des notations graphiques en plus des notations textuelles mais surtout nous avons des types de données sans restriction et un lien fort avec la partie dynamique.

Nous avons également conduit une expérience de génération de code [POI 99] en ActiveJava [AMS 98]. Le comportement dynamique est exprimé avec des STSs ce qui est un avantage important au niveau de la lisibilité par rapport à du CSP. Nous n'avons pas de connecteur de première classe comme en WRIGHT. Nous avons la liberté de définir des connecteurs en KORRIGAN comme des composants particuliers grâce à sa colle très expressive. Finalement l'approche de WRIGHT est limitée car elle considère des systèmes de transitions étiquetés à nombre fini d'états ce qui n'est pas notre cas. Bien sûr ceci pose le problème de la vérification des propriétés dynamiques dans un tel contexte mais diverses réponses sont possibles, notamment [ROY 01] qui s'applique à un sous cadre de KORRIGAN.

Notre approche est voisine de celle d'UML, qui a d'ailleurs inspiré nos notations. Nous suggérons, en effet quand cela est possible, de réutiliser les notations faisant office de quasi-standard actuel. Mais comme notre approche est basée composant nous avons une vue différente de celle d'UML en ce qui concerne les communications et les aspects concurrents. Une autre différence importante est le soucis d'une sémantique formelle [CHO 00] ce qui n'est pas une priorité pour UML. Nous avons des notations spécifiques pour décrire les interfaces dynamiques, les patrons de communication et la concurrence. Le problème de la description des systèmes concurrents est assez délicat en UML. Bien qu'il existe (et sûrement aussi à cause de cela) une grande panoplie de constructions concurrentes il n'est pas évident de structurer un système concurrent de façon lisible et réutilisable. À ce sujet plusieurs auteurs ont signalé le besoin d'un type particulier de diagramme exprimant la structure et les communications d'un système concurrent (on peut voir [MCL 98]). Notre proposition, avec ses vues de composition et de communication, ses interfaces dynamiques et sa colle remédie à ce manque.

KORRIGAN et UML-RT [SEL 98] s'intéressent en partie aux mêmes buts : conception d'architectures, composants dynamiques et réutilisabilité. Toutefois UML-RT intègre le temps-réél et s'utilise au niveau de la conception alors que KORRIGAN s'intéresse aux niveaux spécification (formelle) et conception. Il existe aussi d'autres différences concernant principalement les communications, mais la principale différence est que, contrairement à UML-RT, KORRIGAN propose une notation uniforme pour spécifier les aspects statiques et comportementaux d'un composant.

Enfin, notre approche est voisine des composants actuels (EJB, DCOM, CORBA Components, ...) mais à un niveau formel et non pas conception ou codage. Notre génération de code couvre la majeure partie du langage excepté des mécanismes avancés de synchronisation et l'héritage dynamique. Toutefois nos travaux montrent que nos composants peuvent être rendus exécutables sur un support à objets classique et donc la comparaison avec les composants logiciels actuels devient beaucoup plus pertinente. À l'avantage de ces propositions, nous n'avons pas de distinction `home-remote` interface, notre notion de service de nommage est assez rudimentaire et nous ignorons les aspects sécurité, persistance et réflexivité.

5. Conclusion

Dans nos travaux précédents nous avons défini une approche formelle basée sur les structures de vues pour la spécification des systèmes mixtes, *i.e.* avec à la fois du contrôle et des données. Le langage formel défini, KORRIGAN [POI 00], est basé sur les systèmes de transitions symboliques, les spécifications algébriques et une forme simple de logique temporelle. Il permet de décrire un système complexe d'une façon structurée, lisible et unifiée. Ce modèle dispose d'une sémantique opérationnelle qui exprime la cohérence entre les différents aspects. Nous avons par ailleurs défini une proposition d'environnement, ASK [CHO 01b], pour le développement de nos spécifications KORRIGAN. Un trait important de cet environnement est de mettre en place des traducteurs entre spécifications et code, par exemple traduction de KORRIGAN en SDL, LOTOS et ActiveJava.

Dans cet article nous avons montré que KORRIGAN est bien adapté à la définition de composants formels. À l'aide de l'étude de cas d'un service de téléphonie nous avons illustré différents cas de réutilisation en KORRIGAN. Nous avons des moyens de structurer un ensemble de composants hétérogènes, une forme simple d'héritage, la généricité et la possibilité de décrire des patrons de communication. KORRIGAN propose des interfaces dynamiques claires et des moyens de structurer les systèmes complexes avec des données et du contrôle. Il est également bien adapté à la définition d'architectures de systèmes complexes. KORRIGAN supporte à la fois des notations textuelles et des notations graphiques, ce qui est important pour les outils et la lisibilité. Nous pensons que l'approche suivie par KORRIGAN est une voie viable à long terme pour rationaliser la production de composants logiciels réutilisables et fiables.

Une perspective future est de montrer la génération de code CORBA. Un autre aspect qui mérite d'être amélioré est notre approche de l'héritage. Il existe actuellement plusieurs propositions évitant les anomalies connues de l'héritage. Nous disposons d'une séparation des aspects dynamiques et statiques ce qui doit permettre d'adapter la solution de [MES 93] à ces problèmes.

6. Bibliographie

[AMS 98] ÁLVAREZ L. A., MURILLO J. M., SÁNCHEZ F., HERNÁNDEZ J., « ActiveJava, un modelo de programación concurrente orientado a objeto », *III Jornadas de Ingenería del Software, Murcia, Spain*, 1998.

[ALL 97] ALLEN R., GARLAN D., « A formal basis for architectural connection », *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 3, 1997, p. 213–249.

[BOO 87] BOOCH G., « Composants logiciels réutilisables », *Génie logiciel*, , n° 9, 1987, p. 22-26.

[CHO 00] CHOPPY C., POIZAT P., ROYER J.-C., « A Global Semantics for Views », RUS T., Ed., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, vol. 1816 de LNCS, Springer, 2000, p. 165–180.

[CHO 01a] CHOPPY C., POIZAT P., ROYER J.-C., « Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation », HUSSMANN H., Ed.,

FASE'2001 Fundamental Approaches to Software Engineering, vol. 2029 de *LNCS*, Springer, 2001, p. 124–139.

[CHO 01b] CHOPPY C., POIZAT P., ROYER J.-C., « The KORRIGAN Environment », *Journal of Universal Computer Science*, vol. 7, n° 1, 2001, p. 19–36, Special issue on Tools for System Design and Verification.

[CON 99] CONSORTIUM U., « The OMG Unified Modeling Language Specification, Version 1.3 », rapport, June 1999, `ftp ://ftp.omg.org/pub/docs/ad/99-06-08.pdf`.

[ELL 97] ELLSBERGER J., HOGREFE D., SARMA A., *SDL : Formal Object-Oriented Language for Communicating Systems*, Prentice-Hall, 1997.

[FIS 97] FISCHER C., « CSP-OZ : a combination of Object-Z and CSP », BOWMAN H., DERRICK J., Eds., *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Canterbury, UK, 1997, Chapman & Hall, p. 423–438.

[HOA 72] HOARE C., « Proof of Correctness of Data Representations », *Acta Informatica*, vol. 1, 1972, p. 271-281.

[ISO 89] ISO/IEC, « LOTOS : A Formal Description Technique based on the Temporal Ordering of Observational Behaviour », ISO/IEC n° 8807, 1989, International Organization for Standardization.

[KIC 97] KICZALES G., LAMPING J., MENHDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », AKŞIT M., MATSUOKA S., Eds., *ECOOP'97*, vol. 1241 de LNCS, p. 220–242, Springer, 1997.

[MCL 98] MCLAUGHLIN M. J., MOORE A., « Real-Time Extensions to UML », *Dr. Dobb's Journal of Software Tools*, vol. 23, n° 12, 1998, p. 82, 84, 86–93.

[MES 93] MESEGUER J., « Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming », NIERSTRASZ O., Ed., *Proceedings ECOOP'93*, LNCS 707, Kaiserslautern, Germany, 1993, Springer-Verlag, p. 220–246.

[MEY 88] MEYER B., *Object-oriented Software Construction*, International Series in Computer Science, Prentice Hall, 1988.

[PAP 97] PAPATHOMAS M., HERNÁNDEZ J., MURILLO J. M., SÁNCHEZ F., « Inheritance and expressive power in concurrent object-oriented programming », *LMO'97 Langages et Modèles à Objets*, 1997, p. 45–60.

[POI 99] POIZAT P., CHOPPY C., ROYER J.-C., « From Informal Requirements to COOP : a Concurrent Automata Approach », WING J., WOODCOCK J., DAVIES J., Eds., *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, vol. 1709 de LNCS, Toulouse, France, 1999, Springer-, p. 939–962.

[POI 00] POIZAT P., « KORRIGAN : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes », Thèse de doctorat, Institut de Recherche en Informatique de Nantes, Université de Nantes, 2000.

[ROY 01] ROYER J.-C., « Formal Specification and Temporal Proof Techniques for Mixed Systems », *IEEE Proceedings of the IPDPS Conference, FMPPTA*, San Francisco, USA, 2001.

[SEL 98] SELIC B., RUMBAUGH J., « Using UML for Modeling Complex Real-Time Systems », rapport, 1998, Rational Software Corp.

[SMI 97] SMITH G., « A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems », FITZGERALD J., JONES C. B., LUCAS P., Eds., *Formal Methods Europe (FME'97)*, vol. 1313 de LNCS, Springer, 1997, p. 62–81.